AN ENGINEERING APPROACH TO AUTOMATIC PROGRAMMING

by

STUART H. RUBIN *

(3T4U5FH@CMUVM)

Department of Computer Science

Central Michigan University, Mt. Pleasant, MI 48859

ABSTRACT

An exploratory study of the automatic generation and optimization of symbolic programs using DECOM - a prototypical requirement specification model implemented in pure **LISP** was undertaken. It was concluded, on the basis of this study, that symbolic processing languages such as **LISP** can support a style of programming based upon formal transformation and dependent upon the expression of constraints in an object-oriented environment. Such languages can represent all aspects of the software generation process (including heuristic algorithms for effecting parallel search) as dynamic processes since data and program are represented in a uniform format.

Keywords - Constraint-based programming, object-oriented programming, software automation, transformation

1. INTRODUCTION

Software is currently the major cost in information processing systems. It is estimated that information processing systems will account for 13% of the U.S. GNP in 1990 [9]. Higher level languages are necessary in order to obtain significant improvements in the software automation and support process. They also provide substantial decreases in the time and cost of software development, as well as provide major reductions in the cost and time for maintenance and modification of software. Moreover, higher level languages make the management of the software development activity easier and represent a step in the direction of automatic programming.

* Funding for this project, carried out at NOSC, was provided by the Office of Naval Technology (ONT) Summer Postdoctoral Fellowship Program, Projects Office, ASEE, 11 Dupont Circle, Suite 200, Washington, DC 20036 U.S.A.

2. WHY PURE LISP?

There are three good reasons for choosing a functional language like **LISP**: firstly, functional programs are invariably much shorter, more abstract, and easier to understand than their procedural language counterparts; secondly, pure functional programs are amenable to formal analysis and manipulation, and thirdly they are naturally amenable to implementation on a parallel machine [4]. In addition, functional programs describe the transformation of input values to output values - making it possible to establish properties about them and to transform them into more efficient forms through the apparatus of conventional mathematics.

LISP is the oldest and most widely used symbolic language [13]. In it, a list can contain different types of objects. **LISP** is more flexible than statically typed languages like **PASCAL** and **C** because it supports dynamic typing. In **LISP**, function calls, control structures, and built-in operators have the same syntax - facilitating extensibility. Moreover, **LISP** macro expansion is performed by user-defined functions, thus letting an arbitrary computation compute the result of the expansion [13]. Hence, it follows that **LISP** is an excellent language for implementing a transformational synthesizer.

Pure **LISP** is a universal orthogonal subset of **LISP** composed of basic functions for constructing pairs, lists, and numbers; namely **CAR**, **CDR**, **CONS**, **EQ**, and **ATOM**. It also incorporates the control structures using **COND**, recursion, and functional composition (including some means for function definition). In fact, the pure dialect requires list structures containing only atoms and sublists - without numbers or property lists.

Pure **LISP** is declarative in nature. Thus, it helps to avoid unnecessary sequentiality in a specification, which in turn facilitates introduction of parallelism [2]. This is because the order of evaluating the arguments of a multiple-variable lambda-expression is not defined. Hence, such lambda-expressions are a source of parallelism. Moreover, the referential transparency of the language (i.e., variables are bound to expressions) eliminates the need to access complicated data flow analyses and the rules of Church's lambda-calculus can be used as the basis for transformations that manipulate it [2].

Graham has implemented a database system which stores information in the form of **LISP** lists and responds to queries about the information it has stored [5]. Rubin has characterized *learning* as a process for the compression of information in order to yield knowledge (i.e., theory formation and revision) [12]. Hence, the approach to higher level languages advocated herein extends to higher level data and knowledge bases too. It follows that expert systems (and hence their explanation facilities, the knowledge acquisition bottleneck, and control problems) stand to be enormously and favorably impacted by this technology, since their operation depends upon effectively interfacing with one or more knowledge bases.

3. REUSABLE PROGRAM SPECIFICATIONS

The DOD has invested $300 million in the STARS (Software Technology for Adaptable, Reliable Systems) project to investigate software reuse [10]. NASA recognizes that the United States needs a flight-research facility dedicated to rapid avionics prototyping. The Agency is now developing the Ames-Dryden facility to meet that need through reusable software [3].

A higher form of software reuse is needed to overcome the limitations of code reuse. Software reuse becomes more feasible if program specifications are reused instead of

program code. Hence, program specifications should be formally defined in order that they may undergo automatic and formal correctness-preserving transformations. Note that program specifications conveniently serve the purpose of verification and testing. Finally, Sellis et. al. note that the scale of a transformational system is an important design consideration since future expert database systems will contain knowledge bases of significant size which makes main memory insufficient and the use of a database system a necessity [12].

4. AN ALGEBRAIC APPROACH TO FUNCTIONAL SYNTHESIS

The algebraic transformation method is based upon a collection of theorems which state generic equivalences, i.e., semantic equalities, between classes of functions. Then in a functional program, expressions may be rewritten by more efficient, equivalent expressions which are given by one of these theorems. In this way, the process of transformation becomes that of the identification and application of instances of theorems, and the algebraic approach is therefore particularly conducive to mechanization [4]. Optimization is thus a consequence of some underlying analysis which establishes theorems equating an 'original', user-defined function with a more efficient version. As a general definition of an algebraic approach to specification transformation, suppose that the user has defined a pair of abstract data types å, ß, and the corresponding concrete pair å´, ß´ which provide realizations of å, ß respectively. Then, given any function f: å-->ß, it is desired to synthesize a corresponding function, say f': å´-->ß´, which performs operations on objects of type å´ which are isomorphic to the operations performed by f on corresponding objects of type å. The function f' is then the concrete, implementation version of f that was sought.

Many functions f': å´-->ß´ corresponding to f: å-->ß supplied by the programmer may be synthesized by process of algebraic transformation [4]. Each abstract or complex transformation within a system results in a new lower level subsystem.

Functional synthesis may be applied not only to generating programs but also to constructing other complex objects such as relational database implementations of first-order logic queries, VLSI circuit designs, and detailed plans for robotic vehicles to achieve a set of military reconnaissance goals [8]. Hence, it follows that the pursuit of transformative synthesizers has the potential for very broad impact.

5. RESULTS WITH THE DECOM SYSTEM

The DECOM or program decomposition system is intended to minimize the occurrence of software bugs through the use of a top-down structured approach to software reuse. The current version uses a subset of Common **LISP** as its implementation language. Note that DECOM, version 1, while only a prototype, serves to illustrate the potential of the concept of software reuse through knowledge-based design in an object-oriented environment. It also serves as a model for human learning through the use of function(al) composition.

To begin with, consider the programmer working in an object-oriented environment (i.e., without loss of generality). Let

$$\textbf{((FUNC) (((IN}_1\textbf{) (OUT}_1\textbf{)) ((IN}_2\textbf{) (OUT}_2\textbf{)) ... ((IN}_n\textbf{) (OUT}_n\textbf{))))} \qquad (1)$$

define an arbitrary LISP function which satisfies all of the specified distinct I/O constraints (i.e., at least one pair required). DECOM will take such a specification and through the use of knowledge-based heuristic search and user assistance define a function(al), FUNC, such that it satisfies all specifications.

FUNC may be viewed as a procedural knowledge source representing the compressed declarative information contained in all of its constraining I/O pairs. Moreover, the I/O pairs may be viewed as production rules. It then follows that DECOM functions as a fully general rule-inducing system having demonstrable/provable convergence properties. It is worth noting that if FUNC is defined to be a functional, then a knowledge base segment of optimizing transforms may be inductively generated (and tested). Naturally, these functionals will be maintained as fixed points with respect to the contents of the appropriate optimizing knowledge base segment. Different knowledge base segments are represented by different sublists - that's part of the overall beauty of the scheme.

First, DECOM searches the existing knowledge base segment, shown by (2) below, for an exact match of the I/O specification pairs (where m < the number of concurrent processors). If the knowledge base segment is empty, then proceed to the next step.

$$
\begin{aligned}
&[((\mathrm{FUNC}_1)\ ((\mathrm{IN}_{1,1})\ (\mathrm{OUT}_{1,1}))\ ((\mathrm{IN}_{1,2})\ (\mathrm{OUT}_{1,2}))\ \ldots\ ((\mathrm{IN}_{1,n_1})\ (\mathrm{OUT}_{1,n_1})))\quad (2)\\
&\ ((\mathrm{FUNC}_2)\ ((\mathrm{IN}_{2,1})\ (\mathrm{OUT}_{2,1}))\ ((\mathrm{IN}_{2,2})\ (\mathrm{OUT}_{2,2}))\ \ldots\ ((\mathrm{IN}_{2,n_2})\ (\mathrm{OUT}_{2,n_2})))\\
&\ ((\mathrm{FUNC}_3)\ ((\mathrm{IN}_{3,1})\ (\mathrm{OUT}_{3,1}))\ ((\mathrm{IN}_{3,2})\ (\mathrm{OUT}_{3,2}))\ \ldots\ ((\mathrm{IN}_{3,n_3})\ (\mathrm{OUT}_{3,n_3})))\\
&\qquad .\quad .\quad .\quad .\quad .\quad .\quad .\\
&\qquad .\quad .\quad .\quad .\quad .\quad .\quad .\\
&\qquad .\quad .\quad .\quad .\quad .\quad .\quad .\\
&\ ((\mathrm{FUNC}_m)\ ((\mathrm{IN}_{m,1})\ (\mathrm{OUT}_{m,1}))\ ((\mathrm{IN}_{m,2})\ (\mathrm{OUT}_{m,2}))\ \ldots\ ((\mathrm{IN}_{m,n_m})\ (\mathrm{OUT}_{m,n_m})))]
\end{aligned}
$$

If an exact match of the I/O specifications is found, then FUNC_i is returned as the desired LISP function. If however the knowledge base segment is empty, then the user is asked to specify a reduction(s) (if necessary) and proceed with the component derivation as described above - storing their interrelations in the knowledge base in the form of a "macro"-function(al).

If an exact match cannot be found, then a heuristic means-ends analysis attempts to locate the closest match. The heuristic (a dynamic object) is a search function(s) saved in a knowledge base segment. Note that more than one 'closest' match may be explored in parallel. Alternatively, if no I/O specification pairs in the knowledge base satisfy the defined matching metric, then the case is handled as though the knowledge base were empty.

Now, for each closest match found above, a function FUNC_i is searched for such that it distinctly maps each of the given inputs to a corresponding input, $\mathrm{IN}_{k,n}$, where the single function FUNC_k is known by the knowledge base. That is, the knowledge base attempts to map the specification by process of forward composition (i.e., state-space heuristic search).

Much like a genetic approach [6,7], the current approach entails the use of combinatoric search. However, it surpasses the capabilities of a genetic approach in that the powerful technique of means-ends-analysis is fully utilized. Besides, it should be noted, lest the reader struggle with the question as to which approach to take, that genetic algorithms can

be embedded within DECOM. Again, each given specification pair must be distinctly mapped onto the same $FUNC_k$ if the composition is to be successful (extraneous specification pairs are ignored as in the proof of the program form of the *parametization* or *s-m-n* theorem which underpins most of computability theory [1]). Furthermore, this mapping must be effected by the same $FUNC_i$ (which itself may be a specified composition). Then, the desired function (3) is given by the forward composition $f_k \cdot f_i$

$$(FUNC_k\ (FUNC_i\ (IN))) \tag{3}$$

The use of composition may be extended to an arbitrary level, $f_m \cdot f_{m-1} \cdot \ldots \cdot f_2 \cdot f_1$, subject to the number of available concurrent processors. The application of optimization rules can prune the search tree and/or compress the result.

The mapping of outputs is analogous to the case for inputs - except that here, the desired function (4) is given by the backward or inverse composition $f_k \cdot f_i^{-1}$ (i.e., goal-driven heuristic search)

$$(FUNC_k\ (FUNC_i^{-1}\ (OUT))) \tag{4}$$

where $FUNC_i^{-1}$ maps the outputs as described above for the case of the inputs. Note that

$$(FUNC_i^{-1}\ (OUT)) = (FUNC_k\ (FUNC_i\ (IN))) \tag{5}$$

Moreover, there is no reason that an inverse composition, $f_m^{-1} \cdot f_{m-1}^{-1} \cdot \ldots \cdot f_2^{-1} \cdot f_1^{-1}$, cannot be combined with a forward composition for greater efficiency (i.e., bidirectional heuristic search).

New functions (i.e., functions defined by composition as per above) are saved in the knowledge base if and only if they have accepted optimization or have been manually specified due to failure, for whatever reason, to be the result of composition. (Frequently referenced functions should be copied, in expanded form, into a cache.) This is not unlike case-based reasoning, since the larger the knowledge base, the more likely the matching metric is to succeed. Also, it is clear that the matching metric should be a dynamic object(s), saved in the knowledge base, although this aspect has not yet been explored.

Note that erroneous functions may be pulled from the knowledge base at any time - independent of any other functions. They can subsequently be re-synthesized from the (updated) specifications. Hence, the DECOM system, like a neural net or even a DNA program, exhibits a capability for self-repair.

IN and OUT can specify **LISP** functions since again **LISP** makes no syntactic distinction between program and data. It follows that FUNC can serve as a functional - mapping **LISP** functions, meta-functions, or even entire knowledge base segments, and so on. This is vitally important to the efficient working of the described transformational synthesizer (even on a connection machine) because functions carried as specifications define optimizing rewrite rules. Hence, it is generally more efficient to maintain them in a separate knowledge base segment consisting of fixed-point functionals. Note that function(al)s can be recursively defined using a push-down stack of pending tasks.

6. AN INTRODUCTION TO OPTIMIZING TRANSFORMS

One of the key results pertaining to optimizing transforms is that their effects often enable subsequent optimizations. To see this, first consider the following abstract function sequence (6) and the three associated Type 0 rewrite rules:

```
FUNC:  UVWXYZUVW                                    (6)
  R1:  VW --> X
  R2:  XX --> Z
  R3: Z?Z --> Z
```

A derivation sequence (7) is given by:

$$\text{UVWXYZUVW} \xrightarrow{R1} \text{UXXYZUX} \xrightarrow{R2} \text{UZYZUX} \xrightarrow{R3} \text{UZUX} \qquad (7)$$

Note that optimizing rewrite rules are saved as fixed points with respect to the segment in which they reside. That is, the i^{th} segment is such that for all contained rules, there does not exist a contracting rule, Rj, whose antecedent matches any of the patterns found in the i^{th} segment - itself excluded. Note that the use of the term "fixed point" here applies only so far as a one-step derivation is concerned. It does not contradict the undecidability of the minimalization problem [1].

The question arises as to how many different ways the optimizing rules can be applied and with what result. The above example provided no branching in the derivation tree. However, this is obviously a special case. In general, given Type 0 (i.e., universal program or context sensitive contracting) rewrite rules, a derivation can be arbitrarily long and include multiple applications of the same rewrite rules. What this means in a practical context is that abstract program specifications can, in general, derive an arbitrary number of concrete programs. Providing additional specifications may limit this number if the functions defining sequence is altered with respect to the applicable rewrite rules as a result.

Hence, it becomes necessary, in general, to provide an agenda mechanism to order the potential application of the rewrite rules. This agenda is represented in the form of a meta-rulebase segment. An initial sample meta-rulebase segment (8) for the given rulebase follows:

```
MR1:   U --> R1                                     (8)
MR2:  UX --> R2
MR3:  UZ --> R3
```

Meta-rules are treated the same as ordinary rules and thus are saved as fixed points. Hence, (8) is saved as follows:

```
MR1:  U --> R1                                      (9)
MR2:  APPLY (R1) || X --> R2
MR3:  APPLY (R1) || Z --> R3
```

The principal advantage of the fixed point format is that it is more readily amenable to parametization (such as substituting Rk for R1 above), or in the general case, transformation. This advantage applies to rule, meta-rule, ... , $meta^n$ -rulebase segments alike. Note that the Type 0 characterization of the rewrite rules implies the absence of hierarchy in the $meta^n$ -rulebase, n = 0,1, ... Hence, the distinction between rules and meta-rules is merely an illusion which is well-adapted to the purpose of illustration.

Finally, the concern relates to the acquisition of all manner of rules. A recursive model of EBL [11], although not yet implemented, is proposed which induces (meta) rules from their specifications. This idea is consistent with the methodology presented in this paper and will be formally analyzed in forthcoming works. Specifications are optimization constraints which are discovered in retrospect (such as through the use of backtracking). Good (meta) rules tend to reinforce the discovery mechanism; bad (meta) rules achieve the opposite effect. Again, backtracking is but one discovery mechanism - another is heuristic search. The key point, at least at this level of discussion, is that all effective process are given a uniform representation within the system and hence are equally subject to inductive extension.

7. A SIMPLE EXAMPLE

The above exposition will be concretized here by way of a relatively tivial example serving to illustrate the main points made above. To begin with, assume the existence of the following knowledge base segment:

```
[((CAR (LAMBDA (X))) (NIL NIL) ('(A) A) ('(A B) A))                    (10)
 ((CDR (LAMBDA (X))) (NIL NIL) ('(A) NIL) ('(A B) (B)))]
```

Notice that the constraints are ordered - in this case in order of nondecreasing sublist length - in order to facilitate the search and match process. Also, while the number of constraint pairs has been set at three for each function, it is recalled that the only requirement is that at least one constraint pair be defined for each function - with each function allowed arbitrarily more.

Next, a pair of constraints are specified and the sought after function is initialized to the NIL value:

```
(NIL (NIL NIL) ('(A B) B))                                             (11)
```

Now, the knowledge base segment is heuristically searched in a forward direction (the heuristics may reside in a distinct segment) for a function which distinctly maps each input list in the I/O pairs of the unspecified function to the corresponding input lists of a single function residing in the appropriate knowledge base segment. That is, the image under the operation of the applied function will constitute a suitable preimage under the operation of some known function(al) in the relevant knowledge base segment. In the current instance, the images under the operation of the CDR function are NIL and (B), and the preimages under the operation of composition with the CAR function are NIL and (A) respectively. Hence, the following subgoal is attained:

```
((CDR (LAMBDA (X))) (NIL NIL) ('(A B) (B)))                            (12)
```

Next, the above process is iterated where the images under the operation of the CAR function are found to be NIL and B - satisfying all constraints. Hence, the following is the attained goal as desired:

```
((DEFUN HEADTAIL (X) (CAR (CDR X))) (NIL NIL) ('(A B) B))              (13)
```

Note that all LISP errors are interpreted by convention to be the special atom - NIL, or equivalently, the empty list - ().

8. CONCLUSIONS

It follows from experience with the DECOM system that function(al)s can be automatically induced in an extensible coherent environment through the use of a technique for programming by example. The DECOM system also advances the suggestion that AI and distributed computation are interdependent. These fields are unified through the use of the **LISP** symbolic language - a representational vehicle where the data and the program have the same list structure. Other languages may be employed if translated into a suitable symbolic representation. That is, all programming constructs may be placed in bijective correspondence with pure **LISP** constructs.

The use of constraint-based transformation in an object-oriented programming environment promises to allow for the inductive extension of data and knowledge. It is claimed that only then can a system for automatic programming - that is, one capable of learning - be engineered. This claim follows from the evolutionary approach being equally applicable to all effective processes within the system.

ACKNOWLEDGEMENTS

The author would like to express his gratitude to the ASEE, Irwin Goodman, NOSC, Code 421, Alan Gordon, NOSC, Code 013, John H. Holland, Univ. of Mich., Linwood Sutton, NOSC, Code 411, and Robert Wasilausky, NOSC, Code 411 for their time and respective contributions which served to guide the preparation of this paper. A note of appreciation is also accorded to my colleagues at CMU and my parents and brother.

REFERENCES

1. Arbib, M.A., *A Programming Approach to Computability*, Springer-Verlag, New York, NY, 1982.

2. Biggerstaff, T.J. and Perlis, A.J. (eds.), *Software Reusability Volume I*, Addison-Wesley Publishing Co., New York, New York, 1989.

3. Duke, E.L., Brumbaugh, R.W., and Disbrow, J.D., "A Rapid Prototyping Facility for Flight Research in Advanced Systems Concepts," *Computer*, Vol. 22, No. 5, May 1989, pp. 61-66.

4. Field, A.J. and Harrison, P.G., *Functional Programming*, Addison-Wesley Publishing Company, Inc., Menlo Park, CA, 1988.

5. Graham, P., "A **LISP** Query Compiler," *AI Expert*, Vol. 4, No. 6, June 1989, pp. 21-26.

6. Holland, J.H., "Adaptive Algorithms for Discovering and Using General Patterns in Growing Knowledge-Bases," *International Journal of Policy Analysis and Information Systems*, Vol. 4, No. 3, Plenum Press, New York, 1980, pp. 245-268.

7. Holland, J.H., "Genetic Algorithms and Classifier Systems: Foundations and Future Directions," *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, L. Erlbaum Associates, Hillsdale, New Jersey, 1987, pp. 82-89.

8. Linden, T.A. and Markosian, L.Z., *Transformational Synthesis Using **REFINE***, Tech. Report GH4-116847, Reasoning Systems, Inc., Palo Alto, CA, 1988.

9. Markosian, L., Abraido-Fandino, L., and Katzman, S. *Knowledge-Based Software Engineering Using **REFINE***, Tech. Report, Reasoning Systems, Inc., Palo Alto, CA, 1988.

10. McClure, C., *CASE is Software Automation*, Prentice Hall, Englewood Cliffs, NJ, 1989.

11. Mitchell, T.M., Keller, R.M., and Kedar-Cabelli, S.T., "Explanation-Based Generalization: A Unifying View," *Machine Learning*, Vol. 1, No. 1, 1986, pp. 47-80.

12. Rubin, S.H., "Modeling High-Level Knowledge: A Survey," Submitted to *AI Review*, June 1989.

13. Zorn, B., Ho, K., Larus, J., Semenzato, L., and Hilfinger, P., "Multiprocessing Extensions in **SPUR LISP**," *IEEE Software*, Vol. 6, No. 4, July 1989, pp. 41-49.